REMARKS/ARGUMENTS

A petition to revive the present application from unintentional abandonment and authorization for the necessary fees are included herewith. Claims 1-30 are pending in the present application. Claims 1, 11, and 21 have been amended. Reconsideration of the claims is respectfully requested.

I. 35 U.S.C. § 102, Anticipation

The Examiner has rejected claims 1-30 under 35 U.S.C. § 102(e) as being anticipated by *Copeland* et al., Method and Apparatus for Aggressively Rendering Data in a Data Processing System, United States Patent No. 6,557,076 (published April 29, 2003) (hereinafter, "*Copeland*"). This rejection is respectfully traversed.

I.A As to claims 1-7, 11-17, and 21-27

The Examiner has rejected these claims stating:

> Regarding claim 1, Copeland teaches a method for processing objects within a data processing system in a network, the method comprising:
>
> receiving a request message at a first computing device, wherein the request message comprises a source identifier for a fragment (col. 7, line 66 to col. 8, line 5; col. 9, lines 42-56);
>
> performing a first determination for whether or not the request message has been processed by a second computing device that has a fragment-supporting cache management unit (col. 9, lines 48-55);
>
> receiving a response message at the computing device, wherein the response message comprises the fragment (col. 11, lines 26-42; col. 17, lines 5-17);
>
> performing a second determination for whether or not the fragment is to be cached if the computing device can determine that the second computing device has a fragment-supporting cache management unit (col. 17, lines 5-17); and
>
> performing a third determination for whether or not to cache the fragment based on the first determination and the second determination (col. 13, lines 12-22).

Office action dated July 15, 2005, pp. 3-4.

A prior art reference anticipates the claimed invention under 35 U.S.C. § 102 only if every element of a claimed invention is identically shown in that single reference, arranged as they are in the claims. *In re Bond*, 910 F.2d 831, 832, 15 U.S.P.Q.2d 1566, 1567 (Fed. Cir. 1990). All limitations of the claimed invention must be considered when determining patentability. *In re Lowry*, 32 F.3d 1579, 1582,

32 U.S.P.Q.2d 1031, 1034 (Fed. Cir. 1994). Anticipation focuses on whether a claim reads on the product or process a prior art reference discloses, not on what the reference broadly teaches. *Kalman v. Kimberly-Clark Corp.*, 713 F.2d 760, 218 U.S.P.Q. 781 (Fed. Cir. 1983). In this case, each and every feature of the presently claimed invention is not identically shown in the cited reference, arranged as they are in the claims.

Claim 1 recites:

1. A method for processing objects within a data processing system in a network, the method comprising:

receiving a request message at a first computing device, wherein the request message comprises a source identifier for a fragment;

performing a first determination for whether or not the request message has been processed by a second computing device that has a fragment-supporting cache management unit;

receiving a response message at the first computing device, wherein the response message comprises the fragment ;

performing a second determination for whether or not the fragment is to be cached if the first computing device can determine that the second computing device has a fragment-supporting cache management unit; and

performing a third determination for whether or not to cache the fragment based on the first determination and the second determination.

In the present case, each and every step in claim 1 is not shown in the cited reference. In particular, the steps of performing the first, second and third determinations as recited in claim 1 are not taught by *Copeland*.

The Examiner cites the following section as teaching the step of performing a first determination for whether or not the request message has been processed by a second computing device that has a fragment-supporting cache management unit:

> Fragment **700** also contains a URL **704**, which is the URL relative to the server for this fragment. For a top-level fragment (e.g. a JSP that is externally requested), this could be obtained from the HTTP request object's URL. For a contained fragment, this is the JSP file name URL.
>
> Fragment **700** also contains a cache it **706** indicator, which is metadata identifying whether the fragment should be cached. Cache it **706** allows a runtime decision about whether a particular instance (fragment) of the JSP should be cached. For example, on a product display JSP, it may be known that products from a certain category are either accessed too infrequently or changed too frequently to be cached worthwhile caching.

Copeland, col. 9, ll. 48-55.

This section in *Copeland* describes the structure of a fragment shown in *Copeland's* figure 7, reproduced below. The section shows, inter alia, that one of the contents of a fragment is a "cache it" indicator. The "cache it" indicator indicates to a runtime environment whether the fragment should be cached. *Copeland's* figure 7 is as follows:

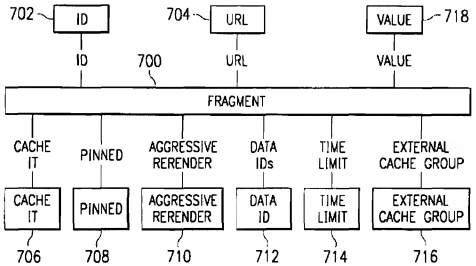


FIG. 7

Copeland, figure 7.

The above figure shows the various contents of a fragment as disclosed by *Copeland*. For fragment 700, the figure shows that the fragment contains a fragment identifier, a uniform resource locator (URL) for the fragment, a value that is the body of the fragment, and several other indicators including the "cache it" indicator.

The section cited by the examiner, and figure 7, corresponding to the cited section, disclose a structure and contents of the structure, not a method. Consequently, the section and figure teach no steps of any method at all. As a result, on its face, the cited section does not teach a step of performing a first determination as recited in claim 1.

Furthermore, the entire disclosure of *Copeland* is devoid of any teaching that a determination should be made that a second data processing system has processed a request. The entire disclosure of *Copeland* is also devoid of a teaching that a determination should also be made whether the second data processing system, not taught, has a fragment-supporting cache management unit. Essentially, the feature "performing a first determination for whether or not the request message has been processed by a second computing device that has a fragment supporting cache-management unit" as recited in claim 1 states that

that these determinations be made with respect to a system not presently handling the request or the fragment in the response. This claim recites receiving a request message at a first computing device, then performing the determination with respect to a second computing device. The first and second computing devices are distinct from each other as identified in claim 1. *Copeland* does not teach a similar determining step involving a system other than the system receiving the request message. Therefore, *Copeland* does not teach, "performing a first determination for whether or not the request message has been processed by a second computing device that has a fragment-supporting cache management unit" as recited in claim 1.

Copeland also does not teach the step of performing the second determination as recited in claim 1 by similar reasoning. This step recites, "performing a second determination for whether or not the fragment is to be cached if the first computing device can determine that the second computing device has a fragment-supporting cache management unit." Because, as reasoned above, *Copeland* does not teach the first determining step involving a second computing device, *Copeland* cannot teach the second determining step, which is dependent on the finding of the first determining step. The examiner cites the following section in *Copeland* as teaching this step of performing a second determination:

> In turn, the response is sent to fragment cache **1520** in WAS JVM **1502** (step **e8**). If a fragment is present in this response. The client cache will receive a fragment from the second level fragment cache if it is available at that second level. If the fragment is found, then the client cache will put the fragment in its cache. This response is returned to JSP engine **1516** (step **e9**). If the response indicated that a fragment was absent, then the JSP is executed. The JSP is executed by calling the JSP's service method (step **e10**). If the fragment was present, the fragment is then returned to the caller requesting the content. More than one fragment might be returned to the caller as a page or in other case, the fragment may make up the entire page.

Copeland, col. 17, ll. 5-17.

As can be seen, this cited section describes the delivery of a fragment from one of two sources in *Copeland's* invention. According to this section request for a fragment is serviced by a web application server's (WAS) Java virtual machine (JVM), by requesting the fragment from a coordinating Java virtual machine (JVM), or executing JSP code. This section describes that when a coordinating JVM does not deliver the needed fragment, the JSP code is executed to generate the fragment. If a fragment is generated, the fragment is returned in response to the request. This is the extent of the teaching present in the cited section. However, the cited section in *Copeland* cannot be fully appreciated in isolation from other parts of this reference. Following additional section provides the information needed to properly understand the teaching of the cited section:

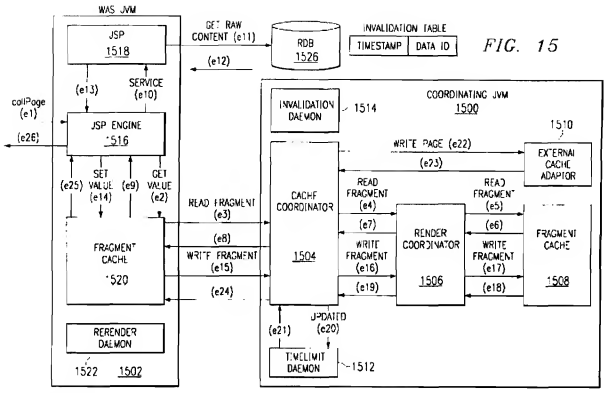
> With reference now to FIG. **15**, a data flow diagram illustrating data flow for

handling an external request using multiple JVMs is depicted in accordance with a preferred embodiment of the present invention. In this example, a coordinating JVM **1500** is present and will interact with Web application server (WAS) JVM **1502**. Coordinating JVM **1500** includes a cache coordinator **1504**, a render coordinator **1506**, a fragment cache **1508**, an external cache adapter **1510**, a time limit daemon **1512**, and an invalidation daemon **1514**. WAS JVM **1502** contains a JSP engine **1516**, JSP **1518**, a fragment cache **1520**, and a render daemon **1522**. In this example, although a single WAS JVM is shown for purposes of describing data flow, coordinating JVM **1500** may handle multiple WAS JVMs in accordance with a preferred embodiment of the present invention.

The execution of a JSP is triggered by an incoming request for a page received in WAS JVM **1502** by JSP engine **1516** (step e**1**). This request in the depicted example is a RequestDispatcher.include( ) request in a HTTP format. JSP engine **1516** then sends a get value call to fragment cache **1520** (step e**2**). In turn, if the fragment needed for the request is not located within the fragment cache **1520**, a read fragment call is sent to cache coordinator **1504** within coordinating JVM **1500** (step e**3**). Cache coordinator **1504** is used as a cache coordinator for all the other JVMs. Cache coordinator **1504** is checked to see if the fragment is present because the fragment may have been rendered by another JVM (not shown) other than WAS JVM **1502**.

Copeland, col. 16, ll. 33-60.

This section of *Copeland* describes the configuration used for providing the fragment in response of a request. The configuration is illustrated in *Copeland's* figure 15, reproduced below. Here, *Copeland* describes that the WAS JVM receives the request for the fragment. WAS JVM passes control to the coordinating JVM in order to receive the requested fragment, if the fragment exists in the coordinating JVM. How the coordinating JVM delivers the fragment, if available, is described elsewhere in *Copeland's* disclosure and is of no consequence here. Essentially, either the coordinating JVM delivers the fragment to the WAS JVM or it does not. If the WAS JVM receives the requested fragment from the coordinating JVM, the fragment is passed in the response to the original request. If the WAS JVM does not receive a fragment from the coordinating JVM, the WAS JVM calls the JSP code to generate the fragment. If the required fragment is generated, the fragment is passed in the response to the original request. Figure 15 in *Copeland* illustrates the interactions described above, and is as follows:



Copeland, figure 15.

This figure shows the relative positions of the WAS JVM, the coordinating JVM, JSP code, and the intervening components. Interactions between these components are illustrated in conformity with the description provided above.

Together, the additional section quoted above and the above figure, do not teach the steps of performing a first and second determinations as claimed. They do not teach these steps because they do not state any such steps or methods in the express disclosure contained therein. Additionally, they do not teach these steps because they do not teach a first and second computing devices in the manner claimed.

Figure 15 does show two JVMs and the additional section quoted above states that more than one WAS JVM can be coordinated through the coordinating JVM. However, persons of ordinary skill in the art know that multiple JVMs can be installed on a single data processing system. Furthermore, persons of ordinary skill in the art also know that multiple instances of the same JVM can be run on a single data processing system. Existence of multiple JVMs on a single data processing system is described in several Java knowledgebases including one published by Sun Microsystems®, which is included in Appendix A of this response. By showing two JVM's *Copeland* does not teach two computing devices as recited in claim 1.

Furthermore, even if, *arguendo*, the two JVMs in *Copeland* are construed to use two computing devices, *Copeland* still fails to show that one JVM performs any determination of the existence of a

fragment-supporting cache management unit in the other JVM. For anticipation, each and every element of the claim must be shown in the reference arranged as they are in the claim. *In re Lowry. Copeland* does not teach the steps of performing the first and second determination as recited in claim 1. Therefore, *Copeland* does not anticipate claim 1.

By the same reasoning, *Copeland* also fails to teach performing a third determination for whether or not to cache the fragment based on the first and second determination as recited in claim 1. *Copeland* teaches a "cache it" indicator for arriving at a decision for caching. However, the methods of making the caching determination are distinct from each other in the recitation of claim 1 and the reference. The claim recites making this determination based on performing the first and second determinations, which *Copeland* does not teach. *Copeland* makes the caching determination based on the "cache it" indicator, which has no bearing on the recitations of the claim. Therefore, *Copeland* further does not teach the step of performing a third determination as recited in claim 1. The examiner cites the following section as teaching this step of claim 1:

> JSP engine **1202** receives a response from JSP **1204** (step **e3**). The JSP engine tries to determine whether the HTML content is cached from the returned response. This determination is made by sending a get value request to fragment cache **1206** (step **e4**). The request for the value is made using the response returned from JSP **1204**. A response is returned by fragment cache **1206** to JSP engine **1202** (step **e5**). If cached content is not present in a response from fragment cache **1206**, JSP **1204** is executed by calling the JSP's service method (step **e6**). Step **e6** occurs without executing step **e4** if the response returned in step **e3** is null.

Copeland, col. 13, ll. 12-22.

This section describes the interactions of the JSP code and the WAS JVM's fragment cache. The section does not teach anything to cure the deficiencies already identified in the *Copeland* disclosure above. A first computing device is not shown as claimed. A second computing device is not shown as claimed. No steps pertaining to a first or second determination are shown as claimed. Because the step of a third determination is based on the first determination and the second determination, the section does not show the step of third determination as well. Therefore, contrary to the examiner's assertion, the contents of this section are insufficient to show the step of performing a third determination as claimed. In addition, as described above, the entire disclosure of *Copeland* fails to teach this step of claim 1.

Independent claims 11 and 21 contain features similar to those present in claim 1. Therefore, *Copeland* does not anticipate claims 11 and 21 by similar reasoning. Claims 2-7, 12-17, and 22-27 depend from one of the independent claims 1, 11, and 21, and are not anticipated by *Copeland* at least by virtue of their dependence from those claims.

I.B **As to claims 8-10, 18-20, and 28-30**

The Examiner has rejected these claims stating:

> Regarding claim 8, Copeland teaches a method for processing objects within a data processing system in a network, the method comprising:
>
> receiving a request message at a server, wherein the request message comprises a source identifier for a fragment (col. 7, line 66 to col. 8, line 5; col. 9, lines 42-56);
>
> generating a response message comprising the fragment (col. 11, lines 26-42; col. 17, lines 5- 1 7); and
>
> inserting in the response message a message header comprising a directive that indicates that the fragment is not to be cached by a first computing device if the first computing device is forwarding the response message to a second computing device that has a fragment-supporting cache management unit (col. 11, line 65 to col. 12, line 35).

Office action dated July 15, 2005, p. 6.

Claim 8 recites:

8. A method for processing objects within a data processing system in a network, the method comprising:

receiving a request message at a server, wherein the request message comprises a source identifier for a fragment;

generating a response message comprising the fragment ; and

inserting in the response message a message header comprising a directive that indicates that the fragment is not to be cached by a first computing device if the first computing device is forwarding the response message to a second computing device that has a fragment-supporting cache management unit.

In the present case, each and every step in claim 8 is not shown in the cited reference. In particular, the steps of inserting in the response message a message header is not shown in *Copeland* as recited in claim 8.

The Examiner cites the following section as teaching this step:

> A Web application server of the present invention includes an internal cache that can also push results of selected pages to external caches. This mechanism allows a decision at a JSP template page granularity to be made as to whether that template would be pushed to an external cache versus an internal cache. Those templates that satisfy the above conditions could exploit the cost-performance advantage of an external cache. Other templates could use the flexibility of the internal cache. In the figures described below, a mechanism for implementing internal and external caches are illustrated in accordance with a preferred embodiment of the present invention. After a page has been rendered, the resulting fragment is pushed to the cache coordinator. The cache coordinator looks at the "externalCacheGroupId" of the fragment. If the cache group is a

> valid cache group defined in this environment, then it will push the fragment to the external caches in that cacheGroup. Mapping of externalCacheGroup ID to a list of all adapters that are a part of that externalCacheGroup. As elements are pushed to the various external caches, some information is maintained to invalidate. This information includes, for example, mapping of fragment data IDs to URLs and mapping of URIs (templates) to URLs. Both are used to invalidate a large number of rendered pages in the external caches depending on either data IDs or their template (URI). FIGS. **12-14** provide illustrations of data flow used in caching dynamic content for Web applications. The examples in FIGS. **12-14** are all described with respect to a single Web application server (WAS) Java virtual machine (JVM). Java™ is an object oriented programming language and environment focusing on defining data as objects and the methods that may be applied to those objects. Java supports only a single inheritance, meaning that each class can inherit from only one other class at any given time. Java also allows for the creation of totally abstract classes known as interfaces, which allow the defining of methods that may be shared with several classes without regard for how other classes are handling the methods.

Copeland, col. 11, l. 65 – col. 12, l. 35.

As shown, the above cited section teaches that a web application server can push a requested page to internal or external caches. After a page has been rendered, the resulting fragment is pushed to a component that coordinates pushing of the fragment to an internal cache or an external cache. The cache coordinator inspects an indicator in the fragment to determine if an identification of a valid external cache is contained therein. The reference calls the indicator "externalCacheGroupID". If the externalCacheGroupID contains a valid identification of an external cache, the cache coordinator pushes the fragment to the identified external cache.

This teaching in the cited section of *Copeland* is different from the step of "inserting in the response message a message header" as recited in claim 8. This section does not teach that the fragment is the response message, or that the externalCacheGroupID is the message header. The section also does not teach that the externalCacheGroupID is "inserted" in the message header as claimed. As a whole, the reference fails to teach the specific features of claim 8, arranged as they are in the claim, as required for anticipation.

Therefore, *Copeland* does not anticipate claim 8. Claims 18 and 28 contain features similar to those in claim 8, and are also not anticipated by *Copeland* by the same reasoning. Claims 9-10, 19-20, and 29-30 are not anticipated by *Copeland* at least by virtue of their dependence from claims 8, 18, and 28. Consequently, *Copeland* does not anticipate claims 1-30, and the rejection of claims 1-30 under 35 U.S.C. § 102(e) has been overcome.

II. Conclusion

It is respectfully urged that the subject application is patentable over *Copeland* and is now in condition for allowance.

The Examiner is invited to call the undersigned at the below-listed telephone number if in the opinion of the Examiner such a telephone conference would expedite or aid the prosecution and examination of this application.

DATE: December 1, 2006

Respectfully submitted,

/Rakesh Garg/

Rakesh Garg
Reg. No. 57,434
Yee & Associates, P.C.
P.O. Box 802333
Dallas, TX 75380
(972) 385-8777
Agent for Applicant

Article

[illegible] JVM

Print-friendly Version

By Janice J. Heiss, March 22, 2005

Some programming tasks related to systems programming cannot be performed solely using the standard Java libraries. Developers might have to use native code, scripting languages, and C or C++ before returning to Java code. For example, stating that a given Java computation should have the exclusive right to use at least 50% of the processor time of its host machine, or that its network usage should be limited to at most 5 MBs, cannot be expressed with the current interfaces.

And, today, when a user runs multiple Java applications concurrently in different [illegible] of the Java virtual machine (JVM), there tends to be substantial duplication of effort in each JVM. For example, when multiple applications are executed concurrently, each has to load, parse, verify, and create runtime representations of all the applications' classes, even such common ones as [illegible]. Application start-up time, memory footprint, and raw execution time are thus negatively impacted. Developers who try to circumvent the problem by using class loaders soon discover that they provide inadequate inter-application isolation and make it difficult to cleanly terminate applications.

Sun Microsystems Laboratories' Project Barcelona research team, led by Dr. Grzegorz Czajkowski, is addressing both of these problems by developing techniques that allow for collocation of multiple applications in a single instance of the virtual machine. Applications are isolated from one another and each "thinks" it has the whole virtual machine all to itself. Aggressive transparent sharing of metadata, such as bytecodes of methods, reduces application footprint and start-up time. Furthermore, through building on the foundation of an isolated Java computation (known as an "isolate"), the team has developed a set of resource management techniques and APIs which allow developers to define new resource types, partition resources among computations, and control resource management overhead.

The Multi-Tasking Virtual Machine

"Across all the metrics of scalability we want to improve the well-being of Java programs."

Grzegorz Czajkowski, Principal Investigator for the Project Barcelona Sun Microsystems Laboratories

The proposed extensions and modifications to the JVM have been prototyped as the Multi-Tasking Virtual Machine (MVM). Existing Java bytecodes can run unmodified, enjoying scalability benefits. New applications can take advantage of new abstractions and interfaces currently missing from the Java platform to accomplish a variety of systems programming tasks. "Across all the metrics of scalability we want to improve the well-being of Java programs," remarks Czajkowski. "We are scaling the Java platform and turning it into a complete operating environment, by adding various APIs and extensions so that anything you currently want to do in the domain of systems programming, you can do from within the language."

The MVM is a general-purpose virtual machine for executing multiple applications written in the Java language, based on the Java HotSpot Virtual Machine and its client compiler. Through hosting multiple tasks, the MVM increases the scalability of the Java platform.

"We went through the JVM runtime and asked the same question about every

component: 'Can this be shared?'"

Grzegorz Czajkowski, Principal Investigator for the Project Barcelona Sun Microsystems Laboratories

"We went through the JVM runtime and asked the same question about every component," explains Czajkowski, "'Can this be shared?' If it could be shared, it was. If it could not, we either replicated it or modified something to end up with maximal sharing." The MVM achieves improved scalability through an aggressive application of its main design principle: transparently share as much of the runtime as possible among applications and replicate only the part of the runtime system that depends on an application state. All of the known APIs and mechanisms of the Java programming language are available to applications. The end result is that one cannot tell if an application is running as an isolate on MVM or on a JVM of its own, barring a faster start-up and lower memory footprint.

While the original MVM prototype targeted the Solaris/SPARC environment, its creators see no major problems in porting it to other platforms. MVM is also a step towards ultimately providing a complete operating environment for Java programs, through a set of carefully designed APIs for resource management, service management, and clustering.

The Isolate API

Java isolates are a key to programming MVM. An isolate is a Java application that shares no objects with other isolates. The Application Isolation API Specification, JSR 121 in the Java Community Process, allows for the dynamic creation and destruction of isolates. A simple example is the creation of an isolate that will execute with a single argument "abc":

The creating isolate may pass contextual information to its offspring that is required by the application that is started within the new isolate. While isolates do not share objects, they can communicate using inter-process communication mechanisms such as sockets, files, or Java Remote Method Invocation (Java RMI). The Isolate API is fully compatible with existing applications and middleware. Applications unaware of the API may be managed as isolates without modification. Similarly, middleware can be unaware of or ignore the API.

"The abstraction of an isolate is important because it is a type-safe language analog of an address space, in which operating system processes are housed," remarks Czajkowski. It offers data protection guarantees, enables clean application termination, and provides a good foundation for per-application resource usage accounting. At the same time, there is no single prescribed way to implement isolates. Some implementations may equate an isolate with an operating system process. Others, much more scalable, will host multiple isolates in a single instance of the virtual machine. The latter can be viewed as a type-safe language kernel, just as, for example, UNIX is a kernel for programs developed in C.

Performance Gains through MVM

MVM offers substantial performance gains. When compared to the HotSpot Virtual Machine, from which it derives, MVM reduces start-up time from 60% to 90%. Thus, a three-second wait for a GUI application is reduced to about one second. Footprint reductions have ranged from a half to a third, while several percent execution time gains were also recorded.

Performance gains derive from two sources. First, because the MVM shares class runtime representations across applications, when one application loads a class, the other can use that class without having to engage in file fetching, parsing, and verifying. And second, with several Java components working in the same process, process switching is avoided when they communicate. For example, MVM collocated a pure Java database and an application server in a single process, which led to an 11 percent increase in throughput, with a request response reduction of 36 percent.

These gains allow certain classes of applications to be practically realized in the Java language. Utilities such as grep, ls, date, and others can now be written in the Java language and executed without slowing start-up.

False Starts in Process Sharing

Before completing the MVM, the Project Barcelona team evaluated two alternative designs which allowed separate JVM processes to share data. In both models, each application executed its own JVM in its own operating system process, while sharing some code with other JVMs.

The first prototype involved sharing meta-data by modifying the Java HotSpot Virtual Machine so that data, classes, and compiled code could be shared through memory. The JVMs collectively maintained a shared-memory area holding runtime representation of loaded classes and dynamically generated native code. "For example," explains Dr. Laurent Daynes of the Project Barcelona team, "an initial JVM would start filling the shared memory area with classes it has loaded and with compiled native code for the methods it most frequently uses. The next JVM arrives and ask if a class it needs is present. If it is, it uses it and replicates and resets those parts that cannot be shared. Optimally, there will be compiled code for some of the methods it needs, and it will not pay any interpretation or dynamic compilation costs for these methods. What the JVM cannot find in the shared memory area, it loads and builds, and then stores it in the shared area for the benefit of all subsequent JVMs. Thus, the third JVM will have an even broader set of classes and compiled methods available, and may run even faster. With this architecture, JVMs collectively build up a shared area of classes and compiled code, and amortize the class loading and dynamic compilation costs."

For the second prototype, the Project Barcelona team encoded entire Java packages as shared libraries so that read-only sections could be automatically shared among JVMs.

Both systems provided mediocre results. Start-up time and memory footprint both decreased by around 10%, numbers that were insufficiently impressive to justify the engineering efforts. And they faced serious engineering challenges involving cross-process synchronization for updating the shared area. "We found that a substantial fraction of the cost and overhead that can be eliminated through sharing across applications actually comes from the existence of multiple processes and multiple runtimes," says Czajkowski.

In addition, Daynes adds, "Whenever a JVM starts up, it has to create a new runtime, with lots of data structures around it, and they actually contribute mostly to the overhead, both in memory and start-up terms."

Transferring the MVM to the Java 2 Platform, Micro Edition (J2ME)

"It is relatively easy to put the MVM in CLDC. It's fast and there is no AWT."

Laurent Daynes, Project Barcelona Sun Microsystems Laboratories

The Project Barcelona team has been collaborating with a group at Sun to build a virtual machine for cell phone devices in an attempt to productize the MVM for the Connected Limited Device Configuration (CLDC). "It is relatively easy to put the MVM in CLDC," observes Daynes. "It's fast and there is no AWT. MVM is a good fit for CLDC for at least two reasons. First, memory is scarce on the types of devices targeted by CLDC, and the ratio of runtime data to application data is typically large, that is, a large fraction of the footprint is taken by the runtime representation of classes. In these circumstances, a JVM that shares most of the runtime data across programs is extremely valuable. Second, CLDC targets a wide range of platforms, some of which have no support for isolation. The ability to launch and control the execution of multiple applications with OS/platform-independent mechanisms increases the portability of both the JVM and applications".

Applying the MVM to the Java 2 Platform, Enterprise Edition (J2EE)

The Java 2 Platform, Enterprise Edition (J2EE) is similar to an operating system in that a J2EE server can host multiple applications. In practice, this is rarely done due to limitations on scalability, weak inter-application isolation, and inadequate resource management facilities in the underlying Java platform. This leads to a proliferation of server [redacted], each typically hosting a single application, with a consequent dramatic increase in the total memory footprint and more complex system administration. The MVM supported multiple [redacted] of the J2EE 1.3.1 Reference Implementation through an efficient

implementation of isolates, thus, substantially increasing scalability, reducing memory footprint, and server startup time when compared with the Java HotSpot Virtual Machine. This was achieved without changing server code and with no loss of performance.

The Resource Management (RM) Interface

While a single J2EE server can support multiple applications, much like a traditional operating system, performance levels may be difficult to control due to the absence of resource management facilities in the Java platform. The Project Barcelona team has created a Resource Management (RM) interface, implemented in the MVM and based on isolates, which offers a flexible, extensible framework for managing resources. The framework is applicable across a broad spectrum, from low-level resources like CPU time, to higher level ones, such as database connections. For example applying the RM API in a J2EE server allows application performance to be controlled flexibly and easily with low overhead and minimal intrusion.

The RM prototype was particularly effective in two ways: First, it eased the retrofitting of existing resource implementations to fit the RM frameworks; and second, it improved the programmable cost/precision tradeoff of the RM API. The prototype is already satisfactory for practical use.

On the Horizon

Work is underway to extend the isolate programming model, the RM, and MVM to clusters of separate machines in an effort to investigate the utility of these mechanisms in large, horizontally scaled, multi-tier J2EE configurations. "Isolates have turned out to have attractive properties as a model for distributed components: the API does not require expressing any collocation preferences, so a two-isolate application should execute unmodified in a single instance of MVM, as well as when each isolate is in a different MVM, on a different computer," explains Czajkowski.

The resource management API naturally extends to the cluster case, due to a strict separation between the decision to consume, or not consume a resource, and the actual implementation of the resource. Another, orthogonal, extension of the APIs is service management. The goal here is to express dependencies between various services in order to automate restart after a given service has gone off-line. Isolates provide a good foundation for defining Java services: they can be asynchronously and independently started and terminated.

MVM Robustness Issues

One challenge facing the MVM parallels a common operating systems problem: A bug in an application brings the application down, but a bug in an operating system brings all applications down. "If there is a problem with an implementation of MVM," admits Czajkowski, "all applications could be affected". This challenge to robustness can be dealt with by running several MVMs, to collocate related computations only. Czajkowski points out that OS kernels get enough debugging attention today so that they are reasonably stable. "The same thing is bound to happen with JVMs," he argues. "Over time, the issue of robustness in MVMs will fade."

Benefits for Users

The scalability improvements may impact users in myriad ways. For example, desktop users will be able to simultaneously run more Java applications due to lowered memory consumption. At the same time, reduced start-up time will significantly improve their experience. Administrators of large computer installations will welcome improved scalability. Developers of middleware infrastructure, such as application servers, as well as enterprise applications, which typically execute on server-class computers or clusters, will also find MVM's programming interfaces attractive. They simplify the task of managing large numbers of components and provide a means to exercise a high level of control over their lifecycle and resource consumption. For example, the Project Barcelona team was able to write a simple Grid system based on isolates and related APIs in four man-months. "I am quite excited about the prospects of writing command line tools in the Java language, or in scripting environments executed on the JVM, such as Groovy or BeanShell," comments Daynes.

The Future of the MVM

The Project Barcelona team is working on a research release of MVM based on the Java HotSpot Virtual Machine, version 1.5. The initial release will contain a version of the Isolate API. Later releases will add the resource management, clustering, and service management APIs. Over time, the members of the Project Barcelona hope to demonstrate that, as Czajkowski puts it, "The Java platform can grow to become a complete operating environment, subsuming the role of modern operating systems as well as middleware platforms."

"The Java platform can grow to become a complete operating environment, subsuming the role of modern operating systems as well as middleware platforms."

Grzegorz Czajkowski, Principal Investigator for the Project Barcelona Sun Microsystems Laboratories

Daynes adds: "Moreover, the Java Platform might not be limited to programs written in the Java language: one can have programs written in various programming languages that target JVM bytecodes, and such applications can co-exist in the same JVM due to MVM's scalable isolation techniques."

Productizing of the MVM itself in the Java 2 Platform, Standard Edition, (J2SE) space is under consideration, but no decisions have yet been made.

In the meantime, the Project Barcelona team is planning the research release of MVM. Please check it out for more information about the release date and contents.

See Also

The Project Barcelona

A Resource Management Interface for the Java Platform

Java Specification Request 121

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

If you would like a reply to your comment, please submit your email address:

Note: We may not respond to all submitted comments.

Submit »